Introduction to Hibernate

21

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- ORM tools.
- Hibernate and how to use Hibernate.
- How to configure database and connect to it.
- Various useful annotations.
- Inheritance mapping.
- How to map Class and Table.
- How to map with set.
- Transactions and caching.
- Hibernate Query Language.
- All about named query.
- Spring Integration.

21.1 | Introduction

Hibernate is one of the most popular and common technologies in Java related projects. It is a framework which acts as an intermediary between a Java application and a database. All the low-level complex work associated with SQL is addressed by Hibernate. Through the addition of Hibernate, the number of lines of code for JDBC can be greatly decreased. The key offering of Hibernate is object-relational mapping (ORM).

What is ORM? As a developer, you have to specify in Hibernate about how an object in your application is mapped to the stored data in the DB. This means that it would map your Java class to any table in the database.

For example, if there is a Java class Employee with four fields:

1. id(int)
2. name(String)
3. designation(String)
4. department(String)

And you have a database table known as Employee.

1. id (INT)
2. name(VARCHAR(45))
3. designation(VARCHAR(45))
4. department(VARCHAR(45))

Then what Hibernate does is that it ensures that this class and table are mapped appropriately. For instance, it can use the object to change the designation of an employee in the table. To generate a Java object in Hibernate, you can write the following code.

```
Employee emp = new Employee ("Albert", "developer", "IT");
```

In order to save it in the DB, you can write the following.

```
int infoId = (Integer) session.save(emp);
```

By doing this, Hibernate is simply running the SQL insert query on the back and saving the data in the DB table. To get or retrieve an object through Hibernate, you can write the following.

```
Employee emp = session.get(Employee.class, infoId);
```

In this code the `session.get` method searches for the class of the object and also takes the primary id of the object (i.e., "infoId"). On the back, Hibernate processes this information by going to the DB and looking for a column with this primary ID.

Now, what to do if you require all of the "Employee" objects? This can be done through Hibernate's support for query. To do this, you can write the following.

```
Query q = session.createQuery("from Employee");
List<Employee> employees = query.list();
```

What the above code does is that it goes in the database table with the name "Employee" and then list all its entries. This is Hibernate Query Language (HQL) at work which would use explain later in our tutorial.

QUICK CHALLENGE Create a chart to show the differences between JDBC approach and Hibernate approach. Also list the advantages and disadvantages of using both.

21.2 | Architecture

Hibernate uses a layered architecture as shown in Figure 21.1. It allows a user to work without explicitly knowing the underlying APIs. Hibernate takes advantage of the configuration and database data which is used to offer persistence services to the application. Hibernate uses JNDI, JDBC, JTA, Java Naming, and other existing Java APIs. It is used to take advantage of a basic abstraction level for relational databases. This means that it is supported by most of the databases.

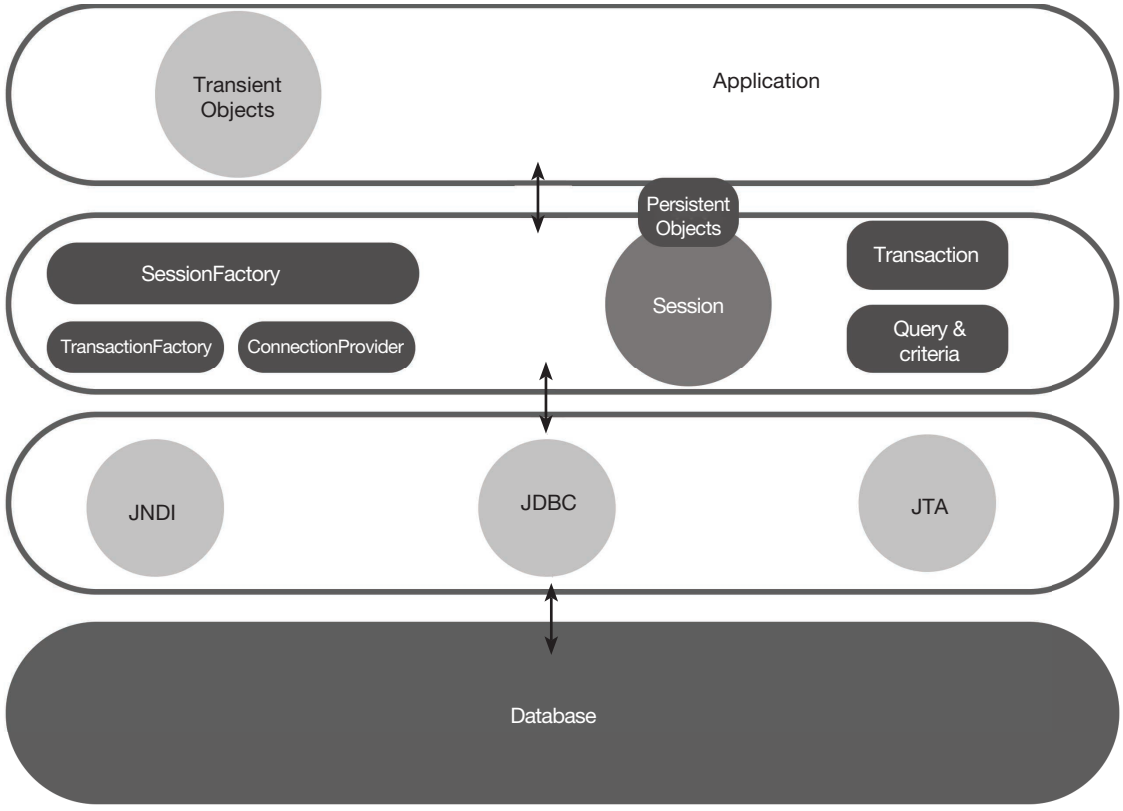

Figure 21.1 Hibernate architecture.

21.2.1 Configuration Object

The first Hibernate object in a Hibernate application is called *configuration object*. It is generated during application initialization. It has details of properties and configuration file. This object has two main components:

1. **Database connection:** This is managed by multiple configuration files. These files are hibernate.cfg.xml and hibernate.properties.
2. **Class mapping setup:** This generates the connection between the DB tables and the Java classes.

Flash Quiz How can you connect multiple ORM tools to Spring MVC?

21.2.2 Session

To establish a physical connection with a database, you need a session. The object of session is lightweight. It is used every time a DB interaction is required. Session objects are used to retrieve persistent objects. The session objects are not allowed to remain open for long periods of time as they are not deemed thread safe. Hence, they are created and destroyed according to the application requirements. The main objective of the session object is to perform read, create, and remove operations on related mapped entity classes instances.

21.2.3 Methods

1. `Transaction beginTransaction():` It starts a unit of work and forwards the relevant Transaction object.
2. `Void cancelQuery():` It terminates the current query execution.
3. `void clear():` It fully clears the session.
4. `connection close():` It ends the session either by cleaning or releasing the connections related to JDBC.
5. `criteria createCriteria(Class persistentClass):` It generates a Criteria instance for a superclass of a specified entity class.
6. `CriteriaCreateCriteria (String entityName):` It generates a fresh Criteria instance to help with the entity name.
7. `Serializable getIdentifier (Object object):` It is used for a given entity's identifier value.

Flash Quiz What is the use of Transaction Management in Hibernate?

21.2.4 Hibernate Criteria

HQL is mostly used to query the database and receive the results. Usually, it is not preferred for deleting or updating values as then it means that the developer has to manage the table associations.

Hibernate Criteria API offers an object-oriented approach, which can be used to query the DB and get the output. It is important to note that the Hibernate Criteria query cannot be used to delete or update Data Definition Language (DDL) statements or other queries. Instead, it is only used for getting DB results through the use of object-oriented model. This API allows the following:

1. Hibernate Criteria API offers Projection, which can be used for `min()`, `max()`, `sum()`, and other aggregate functions.
2. It uses "ProjectList" for getting the specified columns.
3. Hibernate Criteria is used with join queries where it joins multiple tables.
4. Results with conditions can also be fetched by the Hibernate Criteria API.
5. It also offers the `addOrder()` for ordering the output.

QUICK CHALLENGE List the differences between HQL and SQL languages.

21.2.5 Cache

Hibernate Cache is valuable in providing quick performance for applications. It helps to decrease DB queries and thus decreases the application throughput. There are three types of Cache.

1. **First level cache:** This cache is linked with the session object. By default, it is enabled and cannot be turned off. Objects which are cache through such a session are invisible for other sessions. After a session is terminate, all the cache objects are removed.
2. **Second level cache:** By default, it is disabled. To enable it, you have to modify the configuration. Infinispan and EHCache offer the Hibernate second level cache.
3. **Query cache:** Hibernate is used also for caching a query's result set. It does not cache entities' state cache. It is used for caching only the identifier values and the value type results. Hence, it is used with the second-level cache.

How will application behave if no Cache is used?

21.3 | Installation and Configuration

In order to use hibernate, we need to first install database, configure and setup database. In the following sections, we will be installing database, testing database connection using JDBC, and configuring hibernate.

21.3.1 Database

In this chapter, we are using MySQL database for the Hibernate example. Install the latest MySQL Community Server. When you are done installing it, go to Start and search for "Workbench". Open it and connect it to the server instance.

In MySQL, we have created a user "hbstudent", which contains the following columns.

1. Id
2. first_name
3. last_name
4. email

List all the databases that can work with Hibernate.

21.3.2 Eclipse

In Eclipse, create a new "Java Project", we have named ours as "hib". When the project is generated, right click on it and then go to New → Folder. Name the folder as "lib". This folder would be used to store JDBC and Hibernate files. Now, go to Hibernate's official website. Check the latest Hibernate ORM release and download it. After extracting the folder, go in the "required" folder and copy all the files. Come back to Eclipse and paste it in the "lib" folder.

Afterward, open this website and download the latest Connector. After extracting it, copy the JAR file (named like mysql-connector-java) and then paste it in the "lib" folder.

Finally, right click on the project and go to → Properties → Java Build Path → Libraries. Choose "Add Jars" and go to your project and highlight all recently added JAR files. Apply the changes and close the Properties.

21.3.3 JDBC

Before moving forward, it is important to test Java Database Connectivity (JDBC). Create a sample package in the project (we have named it "com.hib.jdbc") and then add a class in it with the following code.

```
import java.sql.Connection;
import java.sql.DriverManager;
public class exampleJDBC {
    public static void main(String args[]) {
        String jdbcUrl = "jdbc:mysql://localhost:3306/hb_student_tracker?useSSL=false";
        String user = "hbstudent";
        String pass = "hbstudent";
        try {
            System.out.println("Attempting to connect to DB: " + jdbcUrl);
            Connection myConn = DriverManager.getConnection(jdbcUrl, user, pass);
            System.out.println("Congratulations! The Connection Is Successful!");
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
    }
}
```

If everything is done correctly, then you we get the following output.

```
"Attempting to connect to DB:
jdbc:mysql://localhost:3306/hb_student_tracker?useSSL=false
Congratulations! The Connection Is Successful!"
```

Till now, we have set up and configured everything required to begin with Hibernate. Now to configure Hibernate, we must first have to create a configuration file. This file instructs Hibernate about the information required to establish a connection with the database. It would contain the JDBC URL, id, password, and other pieces of information. Right click on the "src" of the project and then create a new file with the name "hibernate.cfg.xml". Copy the following code in the file.

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- JDBC Database connection settings -->
        <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/hb_student_tracker?
useSSL=false&amp;serverTimezone=UTC</property>
        <property name="connection.username">hbstudent</property>
        <property name="connection.password">hbstudent</property>
        <!-- JDBC connection pool settings ... using built-in test pool -->
        <property name="connection.pool_size">1</property>
        <!-- Select our SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- Echo the SQL to stdout -->
        <property name="show_sql">true</property>
        <!-- Set the current session context -->
        <property name="current_session_context_class">thread</property>
    </session-factory>
</hibernate-configuration>
```

21.3.4 Annotations

In Hibernate, there is a term known as "Entity Class". It is just a plain old Java object (POJO) which is mapped with a table in the database. This mapping can be done through two options. You can either use the older approach to generate XML configuration files or you can adopt the modern approach to use Annotations.

In Annotations, you have to first map a Java class to a table in the DB and then map its fields to the columns of that DB. To work with Annotations, create a new package in your project and a "Student" class in it. Add the following code in the class.

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name="student")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="first_name")
    private String firstName;

    @Column(name="last_name")
    private String lastName;

    @Column(name="email")
    private String email;

    public Student() { }

    public Student(String firstName, String lastName, String email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    @Override
    public String toString() {
        return "Student [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName
+ ", email=" + email + "]";
    }
}
```

QUICK CHALLENGE Create a list of all the important annotations.

21.4 | Java Objects in Hibernate

In Hibernate, we are going to use a class called SessionFactory. This is the class which goes through the configuration file. It is also responsible for the generation of the session objects. Session factory is generated once and is then used several times in the application. When this class is generated, it then manufactures "sessions".

The "session" class is the wrapper class for JDBC. It is this object which is required for the storage and retrieval of objects. Bear in mind that it has a short lifespan.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import com.hib.hibernate.demo.entity.Student;
public class CreateStudentDemo {
    public static void main(String[] args) {
        SessionFactory factory = new Configuration().configure("hibernate.cfg.xml").
addAnnotatedClass(Student.class).buildSessionFactory();
        Session session = factory.getCurrentSession();
        try {
            System.out.println("Generate an object for a student");
            Student tempStudent = new Student("Robert", "William", "robw123@abc.com");
            session.beginTransaction();
            System.out.println("the student object is being saved");
            session.save(tempStudent);
            session.getTransaction().commit();
            System.out.println("Completed!");
        }
        finally {
            factory.close();
        }
    }
}
```

Now, open your MySQL table and you will see a new row in your table. Change the values in the above code and run it again. Now you will see a new record in the table.

Flash Quiz How can Hibernate manage database security?

21.4.1 Primary Key

Before going further, you must have a basic understanding of primary key (PK). PK is a DB component which is applicable on a column in a table. The main purpose of putting PK on a column is to establish a relationship in relational database. Bear in mind that PK is always unique; this means that you can only assign those columns as PK which cannot have repeated values. For instance, in an employee table, the employee numbers which are unique can be used as PK. It is also important to note that it cannot store NULL values. For instance, if there is an employee "Josh" then to update its data, its employee number "04" can be used in queries where no other employee carries the same employee number. To assign a column as PK, simply write the following line in the SQL query.

```
Primary Key ("emp_no")
```

In Java, you can use GenerationType.AUTO to choose any suitable technique for a specific DB to generate an ID. By using the GenerationType.Identity, primary keys can be assigned with respect to the identity column. The GenerationType.SEQUENCE class processes primary keys according to a DB sequence. Finally, the GenerationType.TABLE processes primary keys with respect to an underlying table in the DB for maintaining uniqueness.

Go to your MySQL Workbench and right click on the table → Alter Table. You can see your columns and the primary key. In our example, we have turned on auto-increment so your AI column will also be ticked.

21.5 | Inheritance Mapping

There are three types of inheritance mapping techniques in Hibernate.

21.5.1 Table Per Hierarchy

In this mapping, you require @Inheritance(strage=InheritanceType.Single_TABLE), @DiscriminatorColumn, and @DiscriminatorValue annotations. This mapping requires only a single table. One more column is generated in the DB's table for calling the class. This table is referred to as the discriminator column. For example, first generate a persistent class which embodies inheritance. To do this, we have produced three classes. First, we have a Student.java class.

```
package com.hib.example;
import javax.persistence.*;
@Entity
@Table(name = "Jones")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(value="student")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "id")
    private int id;
    @Column(name = "name")
    private String name;
    //setters and getters
}
Our second class is the "fulltime_student.java" class.
package com.hib.example;
import javax.persistence.*;

@Entity
@DiscriminatorValue("fulltimestudent")
public class FulltimeStudent extends Student{

    @Column(name="marks")
    private float marks;

    @Column(name="coursenumber")
    private int coursenumber;

    //setters and getters
}
```

Our final class is the parttime_student.java class.

Now, in the project go in the source of the pom.xml file and add the following dependencies. Make sure that they are placed within the <dependencies> tag.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.3.1.Final</version>
</dependency>

<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc14</artifactId>
    <version>10.2.0.4.0</version>
</dependency>
```

Now, you have to open your hibernate.cfg.xml file and add the list of entity classes.

```
?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-5.3.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">system</property>
        <property name="connection.password">jtp</property>
       <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>

        <mapping class="com.ith.mypackage.Student"/>
        <mapping class="com.ith.mypackage.Fulltime_Student"/>
        <mapping class="com.ith.mypackage.Parttime_Student"/>
    </session-factory>
</hibernate-configuration>
```

However, in order to store persistent object, you will require a class. To do this, create a "StoreTest" Java class.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreTest {
    public static void main(String args[])
    {
        StandardServiceRegistry ssr = new StandardServiceRegistryBuilder().configure
("hibernate.cfg.xml").build();
        Metadata meta = new MetadataSources(ssr).getMetadataBuilder().build();
        SessionFactory factory=meta.getSessionFactoryBuilder().build();
        Session session=factory.openSession();
        Transaction t=session.beginTransaction();

         Employee e1=new Employee();
            e1.setName("Gaurav Chawla");

            Regular_Employee e2=new Regular_Employee();
            e2.setName("Vivek Kumar");
            e2.setSalary(50000);
            e2.setBonus(5);

            Contract_Employee e3=new Contract_Employee();
            e3.setName("Arjun Kumar");
            e3.setPay_per_hour(1000);
            e3.setContract_duration("15 hours");

            session.persist(e1);
            session.persist(e2);
            session.persist(e3);

            t.commit();
            session.close();
            System.out.println("success");
    }
}
```

Run this class and you will see the list of students with their names and other details.

21.5.2 Table Per Concrete Class

In Table per Concrete class, a table is generated for each class. This means that in the DB, there is no table which has nullable values. On the other hand, this gives rise to duplication of columns.

In such a mapping, the @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) annotation is used in the parent class, while the @AttributeOverrides annotation is used in any of the subclasses. The former annotation defines the implementation of Table per Concrete class technique while the latter annotation specifies that the attributes of the parent class would be overridden.

First, generate persistent classes for inheritance.

```
import javax.persistence.*;

@Entity
@Table(name = "employee")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)

    @Column(name = "eid")
    private int eid;

    @Column(name = "e_name")
    private String ename;

    //setters and getters
}
```

Now, generate a class for permanent employees.

```
import javax.persistence.*;

@Entity
@Table(name="permanentemployees102")
@AttributeOverrides({
     @AttributeOverride(name="eid", column=@Column(name="eid")),
     @AttributeOverride(name="ename", column=@Column(name="ename"))
})
public class PermanentEmployee extends Employee{

        @Column(name="pay")
        private float pay;

        @Column(name="leaves")
        private int leaves;

        //setters and getters
}
```

For temporary employees, generate the following class.

```
import javax.persistence.*;
@Entity
@Table(name="temporaryemployees")
@AttributeOverrides({
    @AttributeOverride(name="eid", column=@Column(name="eid")),
    @AttributeOverride(name="ename", column=@Column(name="ename"))
})
public class temporary_Employee extends Employee{
    @Column(name="hourly_wage")
    private float hourly_wage;

    @Column(name="services_type ")
    private String services_type;

    public float gethourly_wage() {
        return hourly_wage;
    }
    public void sethourly_wage(float hourlywage) {
        hourly_wage = hourlywage;
    }
    public String getservices_type() {
        return services_type;
    }
    public void setservices_type(String servicestype) {
        services_type = servicestype;
    }
}
```

Now, in the project go in the source of the pom.xml file and add the following dependencies. Make sure that they are placed within the <dependencies> tag.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.3.1.Final</version>
</dependency>

<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc14</artifactId>
    <version>10.2.0.4.0</version>
</dependency>
In the hibernate.cfg.xml file, apply the following mapping.
?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<!-- Generated by MyEclipse Hibernate Tools. -->
<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">system</property>
        <property name="connection.password">abc</property>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>

        <mapping class="com.ith.examples.Employee"/>
        <mapping class="com.ith.examples.PermanentEmployee"/>
        <mapping class="com.ith.examples.Temporary_Employee"/>
    </session-factory>
</hibernate-configuration>
```

Now generate a class StoreData which can save all these objects related to employee tables.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreData {
    public static void main(String[] args) {
        StandardServiceRegistry reg=new StandardServiceRegistryBuilder().configure
("hibernate.cfg.xml").build();
        Metadata meta=new MetadataSources(reg).getMetadataBuilder().build();
        SessionFactory factory=meta.getSessionFactoryBuilder().build();
        Session session=factory.openSession();
        Transaction trans=session.beginTransaction();

        Employee first=new Employee();
        first.setename("Jack");

        PermanentEmployee second=new PermanentEmployee();
        second.setename("Jones");
        second.setpay(4000);
        second.setleaves(8);

        Temporary_Employee third=new Temporary_Employee();
        third.setename("Bill");
        third.sethourly_wage(2000);
        third.setservices_type("IT");

        session.persist(first);
        session.persist(second);
        session.persist(third);

        trans.commit();
        session.close();
        System.out.println("We have completed the operation successfully");
    }
}
```

21.5.3 Table Per Subclass

In table per subclass, tables are generated as persistent classes though they act as primary and foreign keys. This means that it is not possible to have duplicate columns in the table.

In this technique, you require the parent class to have "@Inheritance(strategy=InheritanceType.JOINED)" and the subclasses to have @PrimaryKeyJoinColumn annotations.

First, we have to generate the persistent classes for inheritance. The first "Employee" class contains the following code.

```
import javax.persistence.*;
@Entity
@Table(name = "employee105")
@Inheritance(strategy=InheritanceType.JOINED)
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "eid")
    private int eid;

    @Column(name = "ename")
    private String ename;

    //setters and getters
}
```

In the second class, "PermanentEmployee" we have the following code.

```
import javax.persistence.*;
@Entity
@Table(name="permanentemployee105")
@PrimaryKeyJoinColumn(name="eid")
public class PermanentEmployee extends Employee{

    @Column(name="pay")
    private float pay;
    @Column(name="leaves")
    private int leaves;

    //setters and getters
}
```

In the third class, we have the following code.

```
import javax.persistence.*;

@Entity
@Table(name="temporaryemployee105")
@PrimaryKeyJoinColumn(name="eid")
public class Temporary_Employee extends Employee{

    @Column(name="hourly_wage")
    private float hourly_wage;

    @Column(name="services_type")
    private String services_type;

    //setters and getters
}
```

Now, in the project go in the source of the pom.xml file and add the following dependencies. Make sure that they are placed within the <dependencies> tag.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.3.1.Final</version>
</dependency>

<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc14</artifactId>
    <version>10.2.0.4.0</version>
</dependency>
```

For the configuration file, do the following.

```
?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<!-- Generated by MyEclipse Hibernate Tools. -->
<hibernate-configuration>
    <session-factory>
       <property name="hbm2ddl.auto">update</property>
       <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
       <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
       <property name="connection.username">system</property>
       <property name="connection.password">abc</property>
       <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>

        <mapping class="com.ith.examples.Employee"/>
        <mapping class="com.ith.examples.PermanentEmployee"/>
        <mapping class="com.ith.examples.Temporary_Employee"/>
    </session-factory>
</hibernate-configuration>
```

Now in the final StoreData class, write the following code.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreData {
    public static void main(String[] args) {
        StandardServiceRegistry reg=new StandardServiceRegistryBuilder().
configure("hibernate.cfg.xml").build();
        Metadata meta=new MetadataSources(reg).getMetadataBuilder().build();
        SessionFactory factory=meta.getSessionFactoryBuilder().build();
        Session session=factory.openSession();
        Transaction trans=session.beginTransaction();
        Employee first=new Employee();
        first.setename("Ryan");

        PermanentEmployee second=new PermanentEmployee();
        second.setename("James");
        second.setpay(7000);
        second.setleaves(4);

        Temporary_Employee third=new Temporary_Employee();
        third.setename("Luke");
        third.sethourly_wage(5000);
        third.setservices_type("accounting");

        session.persist(first);
        session.persist(second);
        session.persist(third);

        trans.commit();
        session.close();
        System.out.println("We have completed the operation successfully");
    }
}
```

21.6 | Collection Mapping

Hibernate supports mapping of collection elements in the persistent class. Their types include list, set, map, and collection. The persistent class has to be specified like the following:

```
import java.util.List;

public class problem {
    private int id;
    private String pname;
    private List<String> solutions;

    //getters and setters
}
```

There are numerous sub-elements in the <class> elements for mapping collections.

21.6.1 Mapping List

When the persistent class contains objects in the "list" format, they can be through either annotation or file. For the above class, we can write.

```
<class name="com.iht.problem" table="p100">
          <id name="id">
          <generator class="increment"></generator>
          </id>
          <property name="pname"></property>

          <list name="solutions" table="sol100">
          <key column="pid"></key>
          <index column="type"></index>
          <element column="solution" type="string"></element>
          </list>

          </class>
```

Here, the <key> element is required to specify the table's foreign key with respect to the problem class identifier. To specify the type, the <index> element is required. To specify the collection's element, <element> is used.

In case collection saves objects of another class, it is necessary to specify the types of relationship one-to-many or many-to-many. In such instances, the persistent class resembles the following.

```
import java.util.List;

public class problem {
    private int id;
    private String pname;
    private List<Solution> solutions;

    //getters and setters
}
import java.util.List;
public class Solution {
    private int id;
    private String solution;
    private String posterName;
    //getters and setters
}
```

The mapping file would be similar to the following.

```
<class name="com.ith.problem" table="p100">
          <id name="id">
          <generator class="increment"></generator>
          </id>
          <property name="pname"></property>

          <list name="solutions" >
          <key column="pid"></key>
          <index column="type"></index>
          <one-to-many class="com.ith.problem"/>
          </list>

 </class>
```

The one-to-many relationship here means that a single problem can have multiple solutions.

21.6.2 Key Element

As already explained, the key element is required to place foreign keys in tables. These tables are joined. By default, FK element is nullable. For non-nullable FK, it is important to define attributes which are not null like.

```
<key column="pid" not-null="true"></key>
```

21.6.3 Collection Mapping with Lists

Generate a persistent class.

```
import java.util.List;

public class Problem {
    private int id;
    private String pname;
    private List<String> solutions;

    //getters and setters
}
```

Now create a configuration file named as problem.hbm.xml and apply the following.

```
?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
          "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">

<hibernate-mapping>
 <class name="com.ith.problem" table="p100">
   <id name="id">
     <generator class="increment"></generator>
   </id>
   <property name="pname"></property>

   <list name="solutions" table="sol100">
     <key column="pid"></key>
     <index column="type"></index>
     <element column="solution" type="string"></element>
   </list>
 </class>
</hibernate-mapping>
```

In the configuration file, apply the following.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">user1</property>
        <property name="connection.password">abc</property>
       <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
    <mapping resource="problem.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

To save the data of the problem class, use the StoreData class.

```
import java.util.ArrayList;
import org.hibernate.*;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreData {
     public static void main(String[] args) {
     StandardServiceRegistry reg=new StandardServiceRegistryBuilder().
configure("hibernate.cfg.xml").build();
        Metadata meta=new MetadataSources(reg).getMetadataBuilder().build();

        SessionFactory fac=meta.getSessionFactoryBuilder().build();
        Session session=fac.openSession();

    Transaction ts=session.beginTransaction();

    ArrayList<String> l1=new ArrayList<String>();
    l1.add("Spring is an ecosystem");
    l1.add("Spring is used to create enterprise web applications");

    ArrayList<String> l2=new ArrayList<String>();
    l2.add("Hibernate is an intermediary between database and applications);
    l2.add("Hibernate is used for managing the dependencies of the application");

    problem p1=new problem();
    p1.setPname("What is Spring Framework and why is it used?");
    p1.setSolutions(l1);

    problem p2=new problem();
    p2.setPname("What is Hibernate and why is it used?");
    p2.setSolutions(l2);

    session.persist(p1);
    session.persist(p2);

    ts.commit();
    session.close();
    System.out.println("The operation was successful");
  }
}
```

21.6.4 Mapping with Set

When there is a set object in the persistent class, then mapping can be done through the mapping file's set element. This element does not need an index element. Set differs from the list as it only contains unique values. Generate a persistent class named "Problem" which has the properties such as pname and solutions.

```
import java.util.Set;

public class Problem {
    private int id;
    private String pname;
    private Set<String> solutions;
    //getters and setters
}
```

Now create a configuration file named problem.hbm.xml and apply the following.

```
?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
          "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">

<hibernate-mapping>
 <class name="com.ith.problem" table="p1003">
   <id name="id">
     <generator class="increment"></generator>
   </id>
   <property name="pname"></property>

   <set name="solutions" table="sol1003">
     <key column="pid"></key>
     <index column="type"></index>
     <element column="solution" type="string"></element>
   </sett>
 </class>

</hibernate-mapping>
```

In the configuration file, apply the following.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">user1</property>
        <property name="connection.password">abc</property>
       <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
    <mapping resource="problem.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

To save the data of the problem class, use the StoreData class.

```
import java.util.HashSet;
import org.hibernate.*;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreData {
  public static void main(String[] args) {
     StandardServiceRegistry reg=new StandardServiceRegistryBuilder().configure("hibernate.
cfg.xml").build();
        Metadata meta=new MetadataSources(reg).getMetadataBuilder().build();
        SessionFactory fac=meta.getSessionFactoryBuilder().build();
        Session session=fac.openSession();
    Transaction ts=session.beginTransaction();

    HashSet<String> s1=new HashSet<String>();
    s1.add("Linked list is a data structure");
    s1.add("Linked list is good for memory purposes");

    HashSet<String> s2=new HashSet<String>();
    s2.add("In stack data structure, the last data to be inserted is processed first");
    s2.add("Stack data structure is based upon the real world concept of stacks like
book stacks");

    problem p1=new problem();
    p1.setPname("What is a linked list and why is it used?");
    p1.setSolutions(s1);

    problem p2=new problem();
    p2.setPname("What is Stack data structure and where did the idea come from?");
    p2.setSolutions(s2);

    session.persist(p1);
    session.persist(p2);

    ts.commit();
    session.close();
    System.out.println("The operation was successful");
 }
}
```

21.7 | Mapping with Map

With Hibernate, it is possible to "map" elements of Map along with relational database management systems. The map is a collection which uses indexes on its back. For the problem class, write the following code.

```
import java.util.Map;

public class Problem {
    private int id;
    private String pname,username;
    private Map<String,String> solutions;

    public problem() {}
    public problem(String pname, String username, Map<String, String> solutions) {
        super();
        this.pname = pname;
        this.username = username;
        this.solutions = solutions;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getPname() {
        return pname;
    }
    public void setPname(String pname) {
        this.pname = pname;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public Map<String, String> getSolutions() {
        return solutions;
    }
    public void setSolutions(Map<String, String> solutions) {
        this.solutions = solutions;
    }
}
```

For the problem.hbm.xml file, apply the following.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">
<hibernate-mapping>
    <class name="com.ith.problem" table="problem800">
        <id name="id">
        <generator class="native"></generator>
        </id>
        <property name="name"></property>
        <property name="username"></property>

        <map name="solutions" table="solution800" cascade="all">
        <key column="problemid"></key>
        <index column="solution" type="string"></index>
        <element column="username" type="string"></element>
        </map>
    </class>
</hibernate-mapping>
```

For the hibernate.cfg.xml file, apply the following.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">user1</property>
        <property name="connection.password">abc</property>
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver
</property>
        <mapping resource="problem.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

To save the data and run the application, generate the "StoreData" class and add the following.

```
Import java.util.HashMap;
import org.hibernate.*;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreData {
 public static void main(String[] args) {
     StandardServiceRegistry reg=new StandardServiceRegistryBuilder().configure
("hibernate.cfg.xml").build();
        Metadata meta=new MetadataSources(reg).getMetadataBuilder().build();

        SessionFactory fac=meta.getSessionFactoryBuilder().build();
        Session session=fac.openSession();

    Transaction ts=session.beginTransaction();
    HashMap<String, String> m1=new HashMap<String, String>();
    m1.put("Linked list is a data structure", "Adam");
    m1.put("Linked list is good for memory purposes","Keith");

    HashMap<String, String> m2=new HashMap<String, String>();
    m2.put("In stack data structure, the last data to be inserted is processed first",
"Daniel");
    m2.put("Stack data structure is based upon the real world concept of stacks like
book stacks", "Daniel");
    problem p1=new problem("What is Linked List and why is it used?", "Scully", m1);
    problem p2=new problem("What is Stack?", "Simon", m2);
    session.persist(p1);
    session.persist(p2);

    ts.commit();
    session.close();
    System.out.println("The operation was successful");
  }
}
```

21.7.1 One-to-Many

When the persistent class contains an object in the form of list with the reference for entity, then mapping is performed through the one-to-many association. In these scenarios, one problem can have multiple solutions where each problem has its own details. Such a persistent class would have the following code.

```
import java.util.List;
public class Problem {
    private int id;
    private String pname;  // name of the problem
    private List<Solution> solutions;
}
```

On the other hand, the Solution class can contain details like the name of the solution, poster etc.

```
public class Solution {
    private int id;
    private String solutionname;
    private String poster;  // the one who posted the solution
}
```

In the Problem class, there are entity's references in the list object for which you required to apply one-to-many association.

In the mapping file, apply the following settings.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
     "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
     "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">

     <hibernate-mapping>
          <class name="com.ith.Problem" table="p601">
          <id name="id">
          <generator class="increment"></generator>
          </id>
          <property name="pname"></property>

          <list name="Solutions" cascade="all">
          <key column="pid"></key>
          <index column="type"></index>
          <one-to-many class="com.ith.Solution"/>
          </list>
          </class>

          <class name="com.ith.Solution" table="sol601">
          <id name="id">
          <generator class="increment"></generator>
          </id>
          <property name="solutionname"></property>
          <property name="poster"></property>
          </class>
     </hibernate-mapping>
```

For the configuration file, apply the following settings.

```
?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">abc</property>
        <property name="connection.password">xyz</property>
       <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
    <mapping resource="question.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

To save the data, you can generate the following class.

```
import java.util.ArrayList;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
public class StoreData {
  public static void main(String[] args) {
    StandardServiceRegistry reg= new StandardServiceRegistryBuilder().configure
("hibernate.cfg.xml").build();
    Metadata m=new MetadataSources(reg).getMetadataBuilder().build();
    SessionFactory fact=m.getSessionFactoryBuilder().build();
    Session session=fact.openSession();
    Transaction ts=session.beginTransaction();
    Solution sol1=new Solution();
    sol1.setSolutionname("HTML is a markup language");
    sol1.setPoster("Andrew");

    Solution sol2=new Solution();
    sol2.setSolutionname("HTML is a front-end technology");
    sol2.setPoster("Michael David");
    Solution sol3=new Solution();
    sol3.setSolutionname("Java is used to develop Android applications. It is also used
to create desktop and web applications.");
    sol3.setPoster("Johnny Garcia");
    Solution sol4=new Solution();
    sol4.setSolutionname("Spring Boot minimizes the amount of lines of code which is
usually required in Spring framework. Therefore, it is used for quick projects.");
    sol4.setPoster("Daniel Smith");
    ArrayList<Solution> l1=new ArrayList<Solution>();
    l1.add(sol1);
    l1.add(sol2);
    ArrayList<Solution> l2=new ArrayList<Solution>();
    l2.add(sol3);
    l2.add(sol4);
```

```
    Problem p1=new Problem();
    p1.setPname("What is HTML?");
    p1.setSolutions(l1);

    Problem p2=new Problem();
    p2.setPname("What is SpringBoot and why is this framework used?");
    p2.setSolutions(l2);

    session.persist(p1);
    session.persist(p2);

    ts.commit();
    session.close();
    System.out.println("The operation was completed successfully");
  }
}
```

21.7.2 Transaction

A transaction is used to mark a unit of work. When a single step faces failure then the complete process faces failure. Hibernate provides an interface for transaction. In Hibernate, Sessions are used to call transactions. Following are some of the methods of transactions.

1. **`void begin():`** Initiates a fresh transaction.
2. **`void commit():`** Puts a stop to the unit of work.
3. **`boolean isAlive():`** Scans whether the transaction is dead or alive.
4. **`void setTimeout(int seconds):`** It specifies a time interval when a transaction is initiated due to a call.
5. **`void rollback():`** Rollsbacks the transaction.

It is recommended that whenever an exception is expected in Hibernate, you should attempt at rollbacking the transaction. This strategy ensures that the resources are free. For instance, consider the following code.

```
Session ssn = null;
Transaction ts = null;
try {
    ssn = sessionFactory.openSession();
    ts = ssn.beginTransaction();
    //code for an action
    ts.commit();
}catch (Exception e) {
    e.printStackTrace();
    ts.rollback();
}
finally {
    ssn.close();
}
```

21.8 | Hibernate Query Language

Similar to the database query language SQL, Hibernate has its own query language known as Hibernate Query Language, or HQL in short. HQL is not reliant on the table name of a database. Instead, it makes use of the class name to apply its function. Therefore, it is considered to be independent of database. Hibernate is easy to learn for Java developers. Additionally, it also offers polymorphic queries.

The query interface is an OOP implementation of Hibernate's Query. Query objects can be accessed through the use of `createQuery()` method.

In order to generate data for the existing records, you can write the following code for a persistent class with the name "student". As you can observe, instead of calling a table, HQL is calling for the class through its identifier.

```
Query q = session.createQuery("from Student");//
List l = q.list();
```

To update the details of a user, you can write the following.

```
Transaction ts = ssn.beginTransaction();
Query q=session.createQuery("Update update User set name=:n where id=:i");
q.setParameter("n","Buck");
q.setParameter("i",222);

int sts=q.executeUpdate();
System.out.println(sts);
ts.commit();
```

For deletion, consider the following example.

```
Query q=session.createQuery("delete from Student where id=305");
q.executeUpdate();
```

You can also apply aggregate functions in HQL. For instance to check the average marks,

```
Query query =session.createQuery("select avg(marks) from Student");
List<Integer> l=query.l ();
System.out.println(list.get(0));
```

To check the minimum marks for a student, you can write the following.

```
Query query =session.createQuery("select min(marks) from Student");
```

To check the maximum marks for a student, you can write the following.

```
Query query =session.createQuery("select max(marks) from Student");
```

To check the total number of students, you can write the following.

```
Query query =session.createQuery("select count(id) from Student");
```

To check the total marks of students, you can write the following.

```
Query query =session.createQuery("select sum(marks) from Student");
```

21.8.1 HCQL

To retrieve records according to the customized filter, Hibernate Criteria Query Language (HCQL) is used. It contains an interface for criteria which is composed of several methods to define set criteria for retrieval of information. For example, you can use HCQL to only see those students who have scored more than 80 marks. Similarly, HCQL can be used to identify only those Students who are in their final year.

To receive all the records for HCQL, you can write the following code.

```
Criteria crt=session.createCriteria(Student.class);//
List l=crt.list();
To check only those records which exist between the 30th and 40th position, you can
write the following.
Criteria crt=session.createCriteria(Student.class);
crt.setFirstResult(30);
crt.setMaxResult(40);
List l=crt.list();
```

To check the records on the basis of marks of students in an ascending arrangement, you can write the following.

```
Criteria crt = session.createCriteria(Student.class);
crt.addOrder(Order.asc("marks"));
List l = crt.list()
```

To check records of only those students who have scored more than 75, you can write the following code.

```
Criteria crt = session.createCriteria(Student.class);
crt.add(Restrictions.gt("marks",75));//
List l = crt.list();
```

To get a specific column through projection, you can write the following piece of code.

```
Criteria crt = session.createCriteria(Student.class);
crt.setProjection(Projections.property("department"));
List l = crt.list();
```

21.8.2 Hibernate Named Query

The Hibernate named query is a strategy which is used to run queries through a specific name. It is similar to the DB concept of alias names. To run a single query, you can use the @NameQuery annotation while multiple queries can be handled through @NameQueries.

For instance, consider the following code.

```
import javax.persistence.*;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@NamedQueries(
    {
        @NamedQuery(
        name = "findStudentByName",
        query = "from Student s where s.name = :name"
        )
    }
)

@Entity
@Table(name="st")
public class Student {
    public String toString(){return id+" "+name+" "+marks+" "+subject;}

    int id;
    String name;
    int marks;
    String subject;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)

    //getters and setters
}
```

In the configuration file, you can save details pertaining to username, password, driver class, etc.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 5.3//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
<hibernate-configuration>
   <session-factory>
      <property name="hbm2ddl.auto">update</property>
      <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
      <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
      <property name="connection.username">abc</property>
      <property name="connection.password">xyz</property>
      <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
      <mapping class="com.ith.Student"/>
   </session-factory>
</hibernate-configuration>
```

You can use a FetchInfo class to add your named query.

```
import java.util.*;
import javax.persistence.*;
import org.hibernate.*;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class FetchInfo {
public static void main(String[] args) {
    StandardServiceRegistry reg=new StandardServiceRegistryBuilder().configure("hibernate.
cfg.xml").build();
    Metadata m=new MetadataSources(reg).getMetadataBuilder().build();
    SessionFactory factory=m.getSessionFactoryBuilder().build();
    Session session=factory.openSession();
    TypedQuery q = session.getNamedQuery("findStudentByName");
    q.setParameter("name","Jack");
    List<Student> students=q.getResultList();
    Iterator<Student> i=students.iterator();
while(i.hasNext()){
        Student s=i.next();
        System.out.println(s);
    }
session.close();
    }
}
```

21.9 | Caching

To enhance an application's performance, Hibernate offers caching which uses the cache to pool an object. This is valuable when the same data requires to be processed continuously. There are two types of cache: first and second level cache.

1. The data of first level cache is stored by the Session object and is configured to be turned on by default. The application as a whole does not have access to this cache.
2. The data of second level cache is stored by the SessionFactory object. Unlike the first level cache, this cache can be accessed by the entire application.

The below example generates a persistent class for Student.

```
import javax.persistence.*;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;
@Entity
@Table(name="s101")
@Cacheable
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY)
public class Student {
    @Id
    private int id;
    private String name;
    private float marks;

    public Student() {}
```

```
    public Student(String name, float marks) {
        super();
        this.name = name;
        this.marks = marks;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public float getMarks() {
        return marks;
    }
    public void setMarks(float marks) {
        this.marks = marks;
    }
}
```

In the pom.xml file, add the following dependencies.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.16.Final</version>
</dependency>

<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc14</artifactId>
    <version>10.2.0.4.0</version>
</dependency>

<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>2.10.3</version>
</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>5.2.16.Final</version>
</dependency>
```

In the configuration file, you have to specify the cache.provider_class in the property.

```
?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 5.2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-5.2.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">abc</property>
        <property name="connection.password">xyz</property>
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <property name="cache.use_second_level_cache">true</property>
        <property name="cache.region.factory_class">org.hibernate.cache.ehcache.
EhCacheRegionFactory</property>
        <mapping class="com.ith.Student"/>
    </session-factory>
</hibernate-configuration>
```

To generate the class that can be used for the retrieval of the persistent object, consider the following code.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class FetchInfo {
    public static void main(String[] args) {
        StandardServiceRegistry reg=new StandardServiceRegistryBuilder().
configure("hibernate.cfg.xml").build();
        Metadata m=new MetadataSources(reg).getMetadataBuilder().build();
        SessionFactory fact=m.getSessionFactoryBuilder().build();
        Session ssn=fact.openSession();
        Student s1=(Student)ssn.load(Student.class,202);
        System.out.println(s1.getId()+" "+s1.getName()+" "+s1.getMarks());
        ssn.close();
        Session ssn2=fact.openSession();
        Employee s2=(Student)ssn2.load(Student.class,202);
        System.out.println(s2.getId()+" "+s2.getName()+" "+s2.getMarks());
        ssn2.close();
    }
}
```

21.10 | Spring Integration

Spring is one of the most popular Java web frameworks in the industry. Hibernate is commonly used with Spring applications to build enterprise level web projects. Unlike previous cases where it was necessary to use the hibernate.cfg.xml file, this is not required in the case of Spring. Instead, those details would go in the file named applicationContext.xml.

A HibernateTemplate class comes up with the Spring framework, thereby eliminating the need of configuration, session, BuildSessionFactory, and transactions. For instance, without Spring, Hibernate would be required to look like the following code for a student.

```
Configuration conf=new Configuration();
conf.configure("hibernate.cfg.xml");
SessionFactory fact=conf.buildSessionFactory();
Session ssn=fact.openSession();
Transaction ts=ssn.beginTransaction();
Student s1=new Student(222,"Robert",67);
ssn.persist(s1);
ts.commit();
ssn.close();
```

On the other hand, if you use Hibernate's Template which is provided by Spring, then you can minimize the above code into only two lines.

```
Student s1=new Student(222,"Robert",67);
hibernateTemplate.save(s1);
```

For a more detailed example, use any database to generate a table.

```
CREATE TABLE "ST200"
( "ID" NUMBER(10,0) NOT NULL ENABLE,
   "NAME" VARCHAR2(255 CHAR),
   "MARKS" FLOAT(126),
   PRIMARY KEY ("ID") ENABLE
)
```

Now generate a Student class.

```
public class Student {
   private int id;
   private String name;
   private float marks;
}
```

For the mapping file,

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
   <class name="com.ith.Student" table="ST200">
      <id name="id">
      <generator class="assigned"></generator>
      </id>
      <property name="name"></property>
      <property name="marks"></property>
   </class>
</hibernate-mapping>
```

Use a class which makes use of the HibernateTemplate.

```
import org.springframework.orm.hibernate3.HibernateTemplate;
import java.util.*;
public class StudentDao {
    HibernateTemplate t;
    public void setTemplate(HibernateTemplate t) {
        this.t = t;
    }
    // to store the data of student
    public void saveStudent(Student s){
        t.save(s);
    }
    //to update the data of student
    public void updateStudent(Student s){
        t.update(s);
    }
    //to delete student
    public void deleteStudent(Student s){
        t.delete(s);
    }
    //to get a student through id
    public Student getById(int id){
        Student s=(Student)t.get(Student.class,id);
        return s;
    }
    //to get all the students
    public List<Student> getStudents(){
        List<Student> l=new ArrayList<Student>();
        l=t.loadAll(Student.class);
        return l;
    }
}
```

In the file for applicationContext.xml, you have to add the details of the DB for an object named as BasicDataSource. The use of this object can be seen in another object, LocalSessionFacgtoryBean, which holds data related to the HibernateProperties and mapping Resources. This object is required in the class of HibernateTemplate. Your applicationContext.xml must look like the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"></property>
        <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"></property>
        <property name="username" value="abc"></property>
        <property name="password" value="xyz"></property> </bean>
    <bean id="mysessionFactory" class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"></property>

        <property name="mappingResources">
        <list>
            <value>student.hbm.xml</value> </list> </property>

        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
                <prop key="hibernate.show_sql">true</prop>
            </props> </property> </bean>

    <bean id="template" class="org.springframework.orm.hibernate3.HibernateTemplate">
        <property name="sessionFactory" ref="mysessionFactory"></property> </bean>

    <bean id="d" class="com.ith.StudentDao">
        <property name="template" ref="template"></property> </bean>
</beans>
```

You can then generate an Insert class which makes use of the StudentDao object and calls out the saveStudent method through the object passing of Student class.

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

public class Insert {
   public static void main(String[] args) {
       Resource res=new ClassPathResource("applicationContext.xml");
       BeanFactory fact=new XmlBeanFactory(res);
       EmployeeDao d=(EmployeeDao)fact.getBean("d");
       Student s=new Student();
       s.setId(102);
       s.setName("Mike");
       s.setMarks(77);
       d.saveStudent(s);
   }
}
```

How do we fetch data in case lazy loading is defined?

Summary

In the earlier days, software developers had to struggle a lot to link databases with Java applications. They had to consider an extensive list of factors and therefore were often left puzzled to find a solution. As a consequence, a lot of complex code was generated. Luckily, Hibernate's entry to the scene has changed things considerably.

Today, Hibernate has become an integral part of Java applications. Hence, if you plan to use databases heavily in your application, then do not take the hard way. Simply begin learning Hibernate and easily manage your objects and tables with utmost ease.

In this chapter, we have learned the following concepts:

1. How ORM tools work.
2. Use of hibernate in mapping classes with database tables.
3. How inheritance mapping works.
4. Various annotations to map entity relations.
5. How transactions work with Hibernate.
6. How caching works with Hibernate.
7. How to integrate with Spring.
8. How to fetch record with Hibernate Query Language.
9. How to use named query.

In Chapter 22, we will work on the project we have defined in Chapter 2. We have worked on the front end side of it and now we will work on setting up Spring MVC web services.

Multiple-Choice Questions

1. Which of the following objects is used to generate SessionFactory object in Hibernate?
   (a) Session
   (b) SessionFactory
   (c) Transaction
   (d) Configuration

2. Which of the following is not a fetching strategy?
   (a) Sub-select fetching
   (b) Select fetching
   (c) Join fetching
   (d) Dselect fetching
3. What is the full form of QBC?
   (a) Query By Column
   (b) Query By Code
   (c) Query By Criteria
   (d) Query By Call
4. Which of the following is the file where database table configuration is stored?
   (a) .ora
   (b) .sql
   (c) .hbm
   (d) .dbm
5. Which one of the following is not a valid type of cache?
   (a) First Level
   (b) Second Level
   (c) Third Level
   (d) None of the above

Review Questions

1. What is ORM? What are the benefits of using it?
2. How can Hibernate connect with Spring MVC?
3. How do you define one-to-one relationship?
4. How do you define one-to-many relationship?
5. What are the essential annotations for setting up Hibernate entity on the Spring side?
6. How do you connect database with Hibernate?

Exercises

1. Design a database for managing inventory in a grocery store. Create all the tables and fields for the same and setup hibernate project for the same. Take examples from this chapter.
2. For the above project, create a list of primary and foreign keys that you will map with the variables from the Java class side.

Project Idea

For the Hotel Booking application we have created using Spring MVC in Chapter 20, create a database. Design tables and their relationships based on the model classes and think of the relationship mapping with annotations.

Recommended Readings

1. Christian Bauer, Gavin King, Gary Gregory, and Linda Demichiel. 2017. *Java Persistence with Hibernate, Second Edition*. Manning Publications: New York
2. Joseph B. Ottinger, Jeff Linwood, and Dave Minter. 2016. *Beginning Hibernate: For Hibernate 5*. Apress: New York
3. Yogesh Prajapati and Vishal Ranapariya. 2015. *Java Hibernate Cookbook*. Packt: Birmingham